A (hopefully) gentle guide to the computer implementation of molecular integrals

Joshua Goings

April 28, 2017

In quantum chemistry, we are often interested in evaluating integrals over Gaussian basis functions. Here I am going to take much of the theory for granted and talk about how one might actually implement the integrals for quantum chemistry by using recursion relationships with Hermite Gaussians. Our goal is to have clean, readable code. I'm writing this for the people are comfortable with the mathematics behind the Gaussian integrals, but want to see a readable computer implementation. I'll talk a bit about some computational considerations at the end, but my goal is to convert equations to code. In fact, I've tried to structure the equations and the code in such a way that the two look very similar.

For a very good overview of integral evaluation, please see:

Helgaker, Trygve, and Peter R. Taylor. "Gaussian basis sets and molecular integrals." Modern Electronic Structure (1995).

I will try and follow the notation used in the above reference.

Mathematical preliminaries

Let's start with some of the basics. First, we have our 3D Gaussian functions

$$G_{ijk}(\mathbf{r},\alpha,\mathbf{A}) = x_A^i y_A^j z_A^k \exp(-\alpha r_A^2) \tag{1}$$

with orbital exponent α, electronic coordinates $\mathbf{r}$, origin $\mathbf{A}$, and

$$\mathbf{r}_A = \mathbf{r} - \mathbf{A} \tag{2}$$

also, i,j,k are the angular quantum numbers (e.g. $i=0$ is s-type, $i=1$ is p type, etc.) Cartesian Gaussians are separable in 3D along x,y,z so that

$$G_{ijk}(\mathbf{r},\alpha,\mathbf{A}) = G_i(x,\alpha,A_x)G_j(y,\alpha,A_y)G_k(z,\alpha,A_z) \tag{3}$$

with the 1D Gaussian

$$G_i(x,\alpha,A_x) = (x-A_x)^i \exp(-\alpha(x-A_x)^2). \tag{4}$$

So far, so good. Let's consider the overlap integral of two 1D Gaussians, a and b

$$S_{ab} = \int G_i(x,\alpha,A_x)G_j(x,\beta,B_x)dx \tag{5}$$

$$= \int K_{AB} x_A^i x_B^j \exp(-p x_P^2)dx \tag{6}$$

where we used the Gaussian product theorem so that

$$K_{AB} = \exp(-qQ_x^2), \tag{7}$$
$$Q_x = \alpha A_x - \beta B_x, \tag{8}$$
$$q = \frac{\alpha\beta}{\alpha+\beta}, \tag{9}$$
$$p = \alpha + \beta, \quad \text{and}, \tag{10}$$
$$P_x = \frac{1}{p}\left(\alpha A_x + \beta B_x\right). \tag{11}$$

When using Hermite Gaussians, we can express S_{ab} as

$$S_{ab} = \int \sum_{t=0}^{i+j} E_t^{ij} \Lambda_t dx \tag{12}$$
$$= \sum_{t=0}^{i+j} E_t^{ij} \int \Lambda_t dx \tag{13}$$
$$= \sum_{t=0}^{i+j} E_t^{ij} \delta_{t0} \sqrt{\frac{\pi}{p}} \tag{14}$$
$$= E_0^{ij} \sqrt{\frac{\pi}{p}} \tag{15}$$

where E_t^{ij} are expansion coefficients (to be determined recursively) and Λ_t is the Hermite Gaussian overlap of two Gaussians a and b. It has a simple expression that kills the sum *via* the Kronecker delta δ_{t0}. It can be shown that the expansion coefficients can be defined using the following recursive definitions

$$E_t^{ij} = \frac{1}{2p} E_{t-1}^{i,j-1} - \frac{qQ_x}{a} E_t^{i,j-1} + (t+1) E_{t+1}^{i,j-1}, \tag{16}$$
$$E_t^{ij} = \frac{1}{2p} E_{t-1}^{i-1,j} + \frac{qQ_x}{a} E_t^{i-1,j} + (t+1) E_{t+1}^{i-1,j}, \tag{17}$$
$$E_0^{00} = K_{AB}, \tag{18}$$
$$E_t^{ij} = 0 \quad \text{if} \quad t < 0, \quad \text{or} \quad t > i + j \tag{19}$$

The first equation gives us a way to reduce the index j and the second gives us a way to reduce index i so that we can get to the third equation, which is our base case. The last equation tells us what to do if we go out of index bounds.

Overlap integrals

The first thing we need to do is implement a function `E` which returns our expansion coefficients E_t^{ij}. Aside from angular momentum i and j from the Gaussian functions, we also need the distance between Gaussians Q_x and the orbital exponent coefficients α and β as inputs.

```
def E(i,j,t,Qx,a,b):
    ''' Recursive definition of Hermite Gaussian coefficients.
        Returns a float.
        a: orbital exponent on Gaussian 'a' (e.g. alpha in the text)
        b: orbital exponent on Gaussian 'b' (e.g. beta in the text)
        i,j: orbital angular momentum number on Gaussian 'a' and 'b'
        t: number nodes in Hermite (depends on type of integral,
           e.g. always zero for overlap integrals)
        Qx: distance between origins of Gaussian 'a' and 'b'
    '''
    p = a + b
    q = a*b/p
    if (t < 0) or (t > (i + j)):
        # out of bounds for t
        return 0.0
    elif i == j == t == 0:
        # base case
        return np.exp(-q*Qx*Qx) # K_AB
    elif j == 0:
        # decrement index i
        return (1/(2*p))*E(i-1,j,t-1,Qx,a,b) - \
               (q*Qx/a)*E(i-1,j,t,Qx,a,b)    + \
               (t+1)*E(i-1,j,t+1,Qx,a,b)
    else:
        # decrement index j
        return (1/(2*p))*E(i,j-1,t-1,Qx,a,b) + \
               (q*Qx/b)*E(i,j-1,t,Qx,a,b)    + \
               (t+1)*E(i,j-1,t+1,Qx,a,b)
```

This is simple enough! So for a 1D overlap between two Gaussians we would just need to evaluate E_0^{ij} and multiply it by $\sqrt{\frac{\pi}{p}}$. Overlap integrals in 3D are just a product of the x, y, z 1D overlaps. We could imagine a 3D `overlap` function like so

```
import numpy as np

def overlap(a,lmn1,A,b,lmn2,B):
    ''' Evaluates overlap integral between two Gaussians
        Returns a float.
        a:    orbital exponent on Gaussian 'a' (e.g. alpha in the text)
        b:    orbital exponent on Gaussian 'b' (e.g. beta in the text)
        lmn1: int tuple containing orbital angular momentum (e.g. (1,0,0))
              for Gaussian 'a'
        lmn2: int tuple containing orbital angular momentum for Gaussian 'b'
        A:    list containing origin of Gaussian 'a', e.g. [1.0, 2.0, 0.0]
        B:    list containing origin of Gaussian 'b'
    '''
    l1,m1,n1 = lmn1 # shell angular momentum on Gaussian 'a'
    l2,m2,n2 = lmn2 # shell angular momentum on Gaussian 'b'
    S1 = E(l1,l2,0,A[0]-B[0],a,b,n[0],A[0]) # X
    S2 = E(m1,m2,0,A[1]-B[1],a,b,n[1],A[1]) # Y
    S3 = E(n1,n2,0,A[2]-B[2],a,b,n[2],A[2]) # Z
    return S1*S2*S3*np.power(np.pi/(a+b),1.5)
```

Note that we are using the `NumPy` package in order to take advantage of the definitions of π and the

fractional power to the 3/2. The above two functions `overlap` and `E` are enough to get us the overlap between two Gaussian functions (primitives), but most basis functions are contracted, meaning they are the sum of multiple Gaussian primitives. It is not too difficult to account for this, and we can finally wrap up our evaluation of overlap integrals with a function `S(a,b)` which returns the overlap integral between two *contracted* Gaussian functions.

```
def S(a,b):
    '''Evaluates overlap between two contracted Gaussians
       Returns float.
       Arguments:
       a: contracted Gaussian 'a', BasisFunction object
       b: contracted Gaussian 'b', BasisFunction object
    '''
    s = 0.0
    for ia, ca in enumerate(a.coefs):
        for ib, cb in enumerate(b.coefs):
            s += a.norm[ia]*b.norm[ib]*ca*cb*\
                    overlap(a.exps[ia],a.shell,a.origin,
                    b.exps[ib],b.shell,b.origin)
    return s
```

Basically, this is just a sum over primitive overlaps, weighted by normalization and coefficient. A word is in order for the arguments, however. In order to keep the number of arguments we have to pass into our functions, we have created `BasisFunction` objects that contain all the relevant data for the basis function, including exponents, normalization, etc. A `BasisFunction` class looks like

```
class BasisFunction(object):
    ''' A class that contains all our basis function data
        Attributes:
        origin: array/list containing the coordinates of the Gaussian origin
        shell:  tuple of angular momentum
        exps:   list of primitive Gaussian exponents
        coefs:  list of primitive Gaussian coefficients
        norm:   list of normalization factors for Gaussian primitives
    '''
    def __init__(self,origin=[0.0,0.0,0.0],shell=(0,0,0),exps=[],coefs=[]):
        self.origin = np.asarray(origin)
        self.shell = shell
        self.exps  = exps
        self.coefs = coefs
        self.norm = None
        self.normalize()

    def normalize(self):
        ''' Routine to normalize the basis functions, in case they
            do not integrate to unity.
        '''
        l,m,n = self.shell
        # self.norm is a list of length equal to number primitives
        self.norm = np.sqrt(np.power(2,2*(l+m+n)+1.5)*
                            np.power(self.exps,l+m+n+1.5)/
                            fact2(2*l-1)/fact2(2*m-1)/
                            fact2(2*n-1)/np.power(np.pi,1.5))
```

So, for example if we had a STO-3G Hydrogen 1s at origin (1.0, 2.0, 3.0), we could create a basis function object for it like so

```
myOrigin = [1.0, 2.0, 3.0]
myShell  = (0,0,0) # p-orbitals would be (1,0,0) or (0,1,0) or (0,0,1), etc.
myExps   = [3.42525091, 0.62391373, 0.16885540]
myCoefs  = [0.15432897, 0.53532814, 0.44463454]
a = BasisFunction(origin=myOrigin,shell=myShell,exps=myExps,coefs=myCoefs)
```

Where we used the EMSL STO-3G definition

```
!  STO-3G  EMSL  Basis Set Exchange Library
! Elements                             References
! --------                             ----------
!  H - Ne: W.J. Hehre, R.F. Stewart and J.A. Pople, J. Chem. Phys. 2657 (1969).

****
H     0
S   3   1.00
      3.42525091           0.15432897
      0.62391373           0.53532814
      0.16885540           0.44463454
****
```

So doing `S(a,a) = 1.0`, since the overlap of a basis function with itself (appropriately normalized) is one.

Kinetic energy integrals

Having finished the overlap integrals, we move on to the kinetic integrals. The kinetic energy integrals can be written in terms of overlap integrals

$$T_{ab} = -\frac{1}{2}\left[D^2_{ij}S_{kl}S_{mn} + S_{ij}D^2_{kl}S_{mn} + S_{in}S_{kl}D^2_{mn}\right] \tag{20}$$

where

$$D^2_{ij} = j(j-1)S_{i,j-2} - 2b(2j+1)S_{ij} + 4b^2 S_{i,j+2} \tag{21}$$

For a 3D primitive, we can form a `kinetic` function analogous to `overlap`,

```
def kinetic(a,lmn1,A,b,lmn2,B):
    ''' Evaluates kinetic energy integral between two Gaussians
        Returns a float.
        a:    orbital exponent on Gaussian 'a' (e.g. alpha in the text)
        b:    orbital exponent on Gaussian 'b' (e.g. beta in the text)
        lmn1: int tuple containing orbital angular momentum (e.g. (1,0,0))
              for Gaussian 'a'
        lmn2: int tuple containing orbital angular momentum for Gaussian 'b'
        A:    list containing origin of Gaussian 'a', e.g. [1.0, 2.0, 0.0]
        B:    list containing origin of Gaussian 'b'
    '''
    l1,m1,n1 = lmn1
    l2,m2,n2 = lmn2
    term0 = b*(2*(l2+m2+n2)+3)*\
                           overlap(a,(l1,m1,n1),A,b,(l2,m2,n2),B)
    term1 = -2*np.power(b,2)*\
                          (overlap(a,(l1,m1,n1),A,b,(l2+2,m2,n2),B) +
                           overlap(a,(l1,m1,n1),A,b,(l2,m2+2,n2),B) +
                           overlap(a,(l1,m1,n1),A,b,(l2,m2,n2+2),B))
    term2 = -0.5*(l2*(l2-1)*overlap(a,(l1,m1,n1),A,b,(l2-2,m2,n2),B) +
                  m2*(m2-1)*overlap(a,(l1,m1,n1),A,b,(l2,m2-2,n2),B) +
                  n2*(n2-1)*overlap(a,(l1,m1,n1),A,b,(l2,m2,n2-2),B))
    return term0+term1+term2
```

and for contracted Gaussians we make our final function `T(a,b)`

```
def T(a,b):
    '''Evaluates kinetic energy between two contracted Gaussians
       Returns float.
       Arguments:
       a: contracted Gaussian 'a', BasisFunction object
       b: contracted Gaussian 'b', BasisFunction object
    '''
    t = 0.0
    for ia, ca in enumerate(a.coefs):
        for ib, cb in enumerate(b.coefs):
            t += a.norm[ia]*b.norm[ib]*ca*cb*\
                    kinetic(a.exps[ia],a.shell,a.origin,\
                    b.exps[ib],b.shell,b.origin)
    return t
```

Nuclear attraction integrals

The last one-body integral I want to consider here is the nuclear attraction integrals. These differ from the overlap and kinetic energy integrals in that the nuclear attraction operator $1/r_C$ is Coulombic, meaning we cannot easily factor the integral into Cartesian components x, y, z.

To evaluate these integrals, we need to set up an auxiliary Hermite Coulomb integral $R^n_{tuv}(p, \mathbf{P}, \mathbf{C})$ that handles the Coulomb interaction between a Gaussian charge distribution centered at $\mathbf{P}$ and a nuclei centered

at $\mathbf{C}$. The Hermite Coulomb integral, like its counterpart E_t^{ij}, is defined recursively:

$$R^n_{t+1,u,v} = tR^{n+1}_{t-1,u,v} + X_{PC}R^{n+1}_{tuv} \tag{22}$$
$$R^n_{t,u+1,v} = tR^{n+1}_{t,u-1,v} + Y_{PC}R^{n+1}_{tuv} \tag{23}$$
$$R^n_{t,u,v+1} = tR^{n+1}_{t,u,v-1} + Z_{PC}R^{n+1}_{tuv} \tag{24}$$
$$R^n_{0,0,0} = (-2p)^n F_n(pR^2_{PC}) \tag{25}$$

where $F_n(T)$ is the Boys function

$$F_n(T) = \int_0^1 \exp(-Tx^2)t^{2n}dt \tag{26}$$

which is a special case of the Kummer confluent hypergeometric function, ${}_1F_1(a,b,x)$

$$F_n(T) = \frac{{}_1F_1(n+\frac{1}{2}, n+\frac{3}{2}, -T)}{2n+1} \tag{27}$$

which is convenient for us, since `SciPy` has an implementation of ${}_1F_1$ as a part of `scipy.special`. So for `R` we can code up the recursion like so

```
def R(t,u,v,n,p,PCx,PCy,PCz,RPC):
    ''' Returns the Coulomb auxiliary Hermite integrals
        Returns a float.
        Arguments:
        t,u,v:   order of Coulomb Hermite derivative in x,y,z
                 (see defs in Helgaker and Taylor)
        n:       order of Boys function
        PCx,y,z: Cartesian vector distance between Gaussian
                 composite center P and nuclear center C
        RPC:     Distance between P and C
    '''
    T = p*RPC*RPC
    val = 0.0
    if t == u == v == 0:
        val += np.power(-2*p,n)*boys(n,T)
    elif t == u == 0:
        if v > 1:
            val += (v-1)*R(t,u,v-2,n+1,p,PCx,PCy,PCz,RPC)
        val += PCz*R(t,u,v-1,n+1,p,PCx,PCy,PCz,RPC)
    elif t == 0:
        if u > 1:
            val += (u-1)*R(t,u-2,v,n+1,p,PCx,PCy,PCz,RPC)
        val += PCy*R(t,u-1,v,n+1,p,PCx,PCy,PCz,RPC)
    else:
        if t > 1:
            val += (t-1)*R(t-2,u,v,n+1,p,PCx,PCy,PCz,RPC)
        val += PCx*R(t-1,u,v,n+1,p,PCx,PCy,PCz,RPC)
    return val
```

and we can define our `boys(n,T)` function as

```
from scipy.special import hyp1f1

def boys(n,T):
    return hyp1f1(n+0.5,n+1.5,-T)/(2.0*n+1.0)
```

There are other definitions of the Boys function of course, in case you do not want to use the `SciPy` built-in. Note that `R` requires knowledge of the composite center **P** from two Gaussians centered at **A** and **B**. We can determine **P** using the Gaussian product center rule

$$P = \frac{\alpha A + \beta B}{\alpha + \beta} \tag{28}$$

which is very simply coded up as

```
def gaussian_product_center(a,A,b,B):
    return (a*A+b*B)/(a+b)
```

Now that we have a the Coulomb auxiliary Hermite integrals R^n_{tuv}, we can form the nuclear attraction integrals with respect to a given nucleus centered at **C**, $V_{ab}(C)$, *via* the expression

$$V_{ab}(C) = \frac{2\pi}{p} \sum_{t,u,v}^{\substack{i+j+1,\\ k+l+1,\\ m+n+1}} E^{ij}_t E^{kl}_u E^{mn}_v R^0_{tuv}(p, \mathbf{P}, \mathbf{C}) \tag{29}$$

```
def nuclear_attraction(a,lmn1,A,b,lmn2,B,C):
    ''' Evaluates kinetic energy integral between two Gaussians
        Returns a float.
        a:    orbital exponent on Gaussian 'a' (e.g. alpha in the text)
        b:    orbital exponent on Gaussian 'b' (e.g. beta in the text)
        lmn1: int tuple containing orbital angular momentum (e.g. (1,0,0))
              for Gaussian 'a'
        lmn2: int tuple containing orbital angular momentum for Gaussian 'b'
        A:    list containing origin of Gaussian 'a', e.g. [1.0, 2.0, 0.0]
        B:    list containing origin of Gaussian 'b'
        C:    list containing origin of nuclear center 'C'
    '''
    l1,m1,n1 = lmn1
    l2,m2,n2 = lmn2
    p = a + b
    P = gaussian_product_center(a,A,b,B) # Gaussian composite center
    RPC = np.linalg.norm(P-C)

    val = 0.0
    for t in xrange(l1+l2+1):
        for u in xrange(m1+m2+1):
            for v in xrange(n1+n2+1):
                val += E(l1,l2,t,A[0]-B[0],a,b) * \
                       E(m1,m2,u,A[1]-B[1],a,b) * \
                       E(n1,n2,v,A[2]-B[2],a,b) * \
                       R(t,u,v,0,p,P[0]-C[0],P[1]-C[1],P[2]-C[2],RPC)
    val *= 2*np.pi/p
    return val
```

And, just like all the other routines, we can wrap it up to treat contracted Gaussians like so:

```
def V(a,b,C):
    '''Evaluates overlap between two contracted Gaussians
       Returns float.
       Arguments:
       a: contracted Gaussian 'a', BasisFunction object
       b: contracted Gaussian 'b', BasisFunction object
       C: center of nucleus
    '''
    v = 0.0
    for ia, ca in enumerate(a.coefs):
        for ib, cb in enumerate(b.coefs):
            v += a.norm[ia]*b.norm[ib]*ca*cb*\
                    nuclear_attraction(a.exps[ia],a.shell,a.origin,
                    b.exps[ib],b.shell,b.origin,C)
    return v
```

Important: Note that this is the nuclear repulsion integral contribution from an atom centered at **C**. To get the full nuclear attraction contribution, you must sum over all the nuclei, as well as scale each term by the appropriate nuclear charge!

Two electron repulsion integrals

We are done with the necessary one-body integrals (for a basic Hartree-Fock energy code, at least) and are ready to move on to the two-body terms: the electron-electron repulsion integrals. Thankfully, much of the work has been done for us on account of the nuclear attraction one-body integrals.

In terms of Hermite integrals, to evaluate the two electron repulsion terms, we must evaluate the summation

$$g_{abcd} = \frac{2\pi^{5/2}}{pq\sqrt{p+q}} \sum_{t,u,v}^{\substack{i+j+1,\\ k+l+1,\\ m+n+1}} E_t^{ij} E_u^{kl} E_v^{mn} \sum_{\tau,\nu,\phi}^{\substack{i'+j'+1,\\ k'+l'+1,\\ m'+n'+1}} E_\tau^{i'j'} E_\nu^{k'l'} E_\phi^{m'n'} (-1)^{\tau+\nu+\phi} R^0_{t+\tau,u+\nu,v+\phi}(p,q,\mathbf{P},\mathbf{Q}) \tag{30}$$

which looks terrible and it is. However, recalling that $p = \alpha + \beta$ letting $q = \gamma + \delta$ (that is, the Gaussian exponents on a and b, and c and d), we can write the equation in a similar form to the nuclear attraction integrals

```
def electron_repulsion(a,lmn1,A,b,lmn2,B,c,lmn3,C,d,lmn4,D):
    ''' Evaluates kinetic energy integral between two Gaussians
        Returns a float.
        a,b,c,d:   orbital exponent on Gaussian 'a','b','c','d'
        lmn1,lmn2
        lmn3,lmn4: int tuple containing orbital angular momentum
                   for Gaussian 'a','b','c','d', respectively
        A,B,C,D:   list containing origin of Gaussian 'a','b','c','d'
    '''
    l1,m1,n1 = lmn1
    l2,m2,n2 = lmn2
    l3,m3,n3 = lmn3
    l4,m4,n4 = lmn4
    p = a+b # composite exponent for P (from Gaussians 'a' and 'b')
    q = c+d # composite exponent for Q (from Gaussians 'c' and 'd')
    alpha = p*q/(p+q)
    P = gaussian_product_center(a,A,b,B) # A and B composite center
    Q = gaussian_product_center(c,C,d,D) # C and D composite center
    RPQ = np.linalg.norm(P-Q)

    val = 0.0
    for t in xrange(l1+l2+1):
        for u in xrange(m1+m2+1):
            for v in xrange(n1+n2+1):
                for tau in xrange(l3+l4+1):
                    for nu in xrange(m3+m4+1):
                        for phi in xrange(n3+n4+1):
                            val += E(l1,l2,t,A[0]-B[0],a,b) * \
                                   E(m1,m2,u,A[1]-B[1],a,b) * \
                                   E(n1,n2,v,A[2]-B[2],a,b) * \
                                   E(l3,l4,tau,C[0]-D[0],c,d) * \
                                   E(m3,m4,nu ,C[1]-D[1],c,d) * \
                                   E(n3,n4,phi,C[2]-D[2],c,d) * \
                                   np.power(-1,tau+nu+phi) * \
                                   R(t+tau,u+nu,v+phi,0,\
                                       alpha,P[0]-Q[0],P[1]-Q[1],P[2]-Q[2],RPQ)

    val *= 2*np.power(np.pi,2.5)/(p*q*np.sqrt(p+q))
    return val
```

And, for completeness' sake, we wrap the above to handle contracted Gaussians

```
def ERI(a,b,c,d):
    '''Evaluates overlap between two contracted Gaussians
       Returns float.
       Arguments:
       a: contracted Gaussian 'a', BasisFunction object
       b: contracted Gaussian 'b', BasisFunction object
       c: contracted Gaussian 'b', BasisFunction object
       d: contracted Gaussian 'b', BasisFunction object
    '''
    eri = 0.0
    for ja, ca in enumerate(a.coefs):
        for jb, cb in enumerate(b.coefs):
            for jc, cc in enumerate(c.coefs):
                for jd, cd in enumerate(d.coefs):
                    eri += a.norm[ja]*b.norm[jb]*c.norm[jc]*d.norm[jd]*\
                           ca*cb*cc*cd*\
                           electron_repulsion(a.exps[ja],a.shell,a.origin,\
                                              b.exps[jb],b.shell,b.origin,\
                                              c.exps[jc],c.shell,c.origin,\
                                              d.exps[jd],d.shell,d.origin)
    return eri
```

And there you have it! All the integrals necessary for a Hartree-Fock SCF code.

Computational efficiency considerations

Our goal here has been to eliminate some of the confusion when it comes to connecting mathematics to actual computer code. So the code that we have shown is hopefully clear and looks nearly identical to the mathematical equations they are supposed to represent. This is one reason we chose `Python` as the code of choice to implement the integrals. It emphasizes readability.

If you use the code as is, you'll find that you can only really handle small systems. To that end, I'll give a few ideas on how to improve the integral code to actually be usable.

First, I would recommend not using pure `Python`. The problem is that we have some pretty deep loops in the code, and nested `for` loops will kill your speed if you insist on sticking with `Python`. Now, you can code in another language, but I would suggest rewriting some of the lower level routines with `Cython` (`http://www.cython.org`). `Cython` statically compiles your code to eliminate many of the `Python` calls that slow your loops down. In my experience, you will get several orders of magnitude speed up. This brings me to my second point. One of the problems with `Python` is that you have the global interpreter lock (GIL) which basically means you cannot run things in parallel. Many of the integral evaluation routines would do well if you could split the work up over multiple CPUs. If you rewrite some of the low level routines such that they do not make `Python` calls anymore, you can turn off the GIL and wrap the code in `OpenMP` directives, or even use `Cython`'s `cython.parallel` module. This will take some thought, but can definitely be done.

A couple other thoughts: be sure to exploit the permutational symmetry of the integrals. The two electron repulsion integrals, for example, can be done in 1/8 of the time just by exploiting these symmetries, which are unrelated to point group. Also, you can exploit many of the integral screening routines, since many of the two electron integrals are effectively zero. There are a lot of tricks out there in the literature, go check them out!